COMPUTER ARCHITECTURE

HW3 :: Pipelined MIPS

By using C programming

모바일시스템공학과 손보경 (32212190)

ssg020910@naver.com

2023.06.21.

목 차

1. Introduction

파이프라인은 작업을 여러 단계로 나누어 병렬로 실행하고 결과를 전달하는 방식이다. 이는 마이크로아키텍처가 각 단계에서 다른 명령어를 처리하는 것을 의미한다. 명령어는 프로그램 순서대로 연속적인 단계에서 처리된다.

이 보고서에서는 파이프라인 마이크로아키텍처의 작동, 구현, 그리고 프로젝트의 평가가 담겨 있다. 파이프라인 제작은 MIPS ISA를 사용하는 마이크로아키텍처를 구현하고 최적화하기 위한 큰 프로젝트의 마지막 부분이다. 여기서 성능을 향상시키기 위해 이전의 싱글 사이클 마이크로아키텍처에 고급 개념을 적용할 것이다.

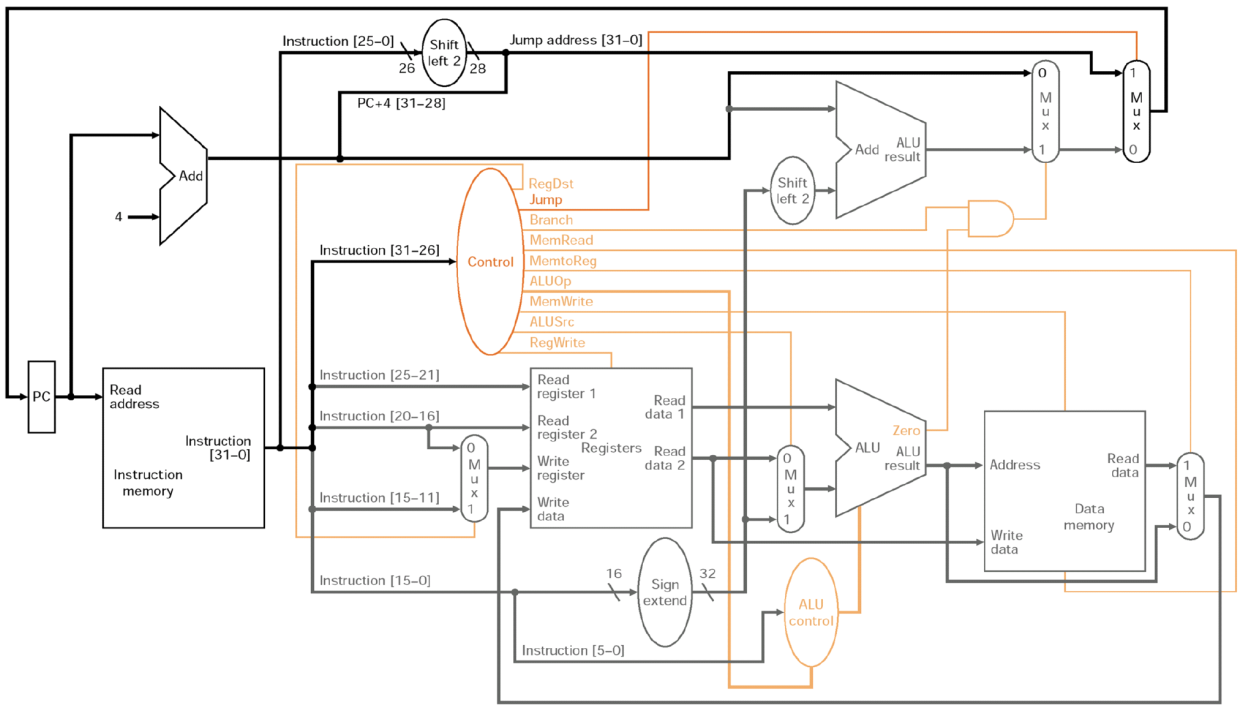
2. Requirements

Index	Requirement
1	Make a MIPS CPU emulator
2	Before the execution, the binary file is loaded into the memory a) Initial value of register R31 is 0xFFFFFFFF b) Initial value of register R29 is 0x10000000 c) Initial value of other registers are 0x0000:0000
3	MIPS executes instructions in the following stages a) Instruction fetch b) Instruction decode c) Execution d) Load/store result to memory e) Write back to reg. file
4	We assume that each stage takes one cycle
5	Cycle is updated at the end of instruction execution loop
6	Pipeline with data hazards a) Detect and wait b) Detect and forward/bypass (with option, extra point)
7	Pipeline with control hazards a) Detect and wait b) Static branch prediction (with option, extra point) c) Dynamic branch prediction (with option, extra point)

3. Concepts

파이프라인 MIPS 시뮬레이터를 들어가기 전에, 주로 구현에 사용되는 개념인 싱글 사이클, 멀티 사이클, 파이프라인, Hazard, Datapath에 대해 간략히 설명하겠다.

3-1. Single-Cycle Microarchitecture



싱글사이클(single-cycle)은 컴퓨터 아키텍처에서 사용되는 명령어 처리 방법 중 하나다. 이 방법은 하나의 명령어를 처리하는 데에 전체 주기를 할당하는 방식으로 동작한다. 각 명령어는 단일 주기 안에서 모든 단계를 거쳐 완전히 처리된다. CPU는 내부 회로를 동작시키기 위해 일정한 주기로 규칙적인 전기 신호를 발생시키는데 이를 Clock이라고 한다. 즉 하나의 클럭 사이클은 한 번의 전기 신호를 말하고, 한 클럭에 걸리는 시간을 클럭 주기라고 한다. 싱글 사이클 프로세서는 그 사이클 동안 하나의 명령어를 처리하는 프로세서를 의미한다. 하지만 싱글사이클 방식은 모든 명령어에 동일한 시간을 할당하기 때문에, 명령어의 복잡성과 실행 시간이 다른 경우에는 자원의 낭비가 발생할 수 있다. 예를 들어 명령어는 각각 사이클 시간이 달라서 제일 긴 사이클 시간을 가진 명령어를 기준으로 사이클 시간을 설정한다면, 처리 시간의 낭비가 발생한다. 또한, 싱글 사이클을 실행하면 하드웨어의 활용도도 낮다. 이런 문제를 보완하기 위해서 멀티 사이클이 등장하게 되었다.

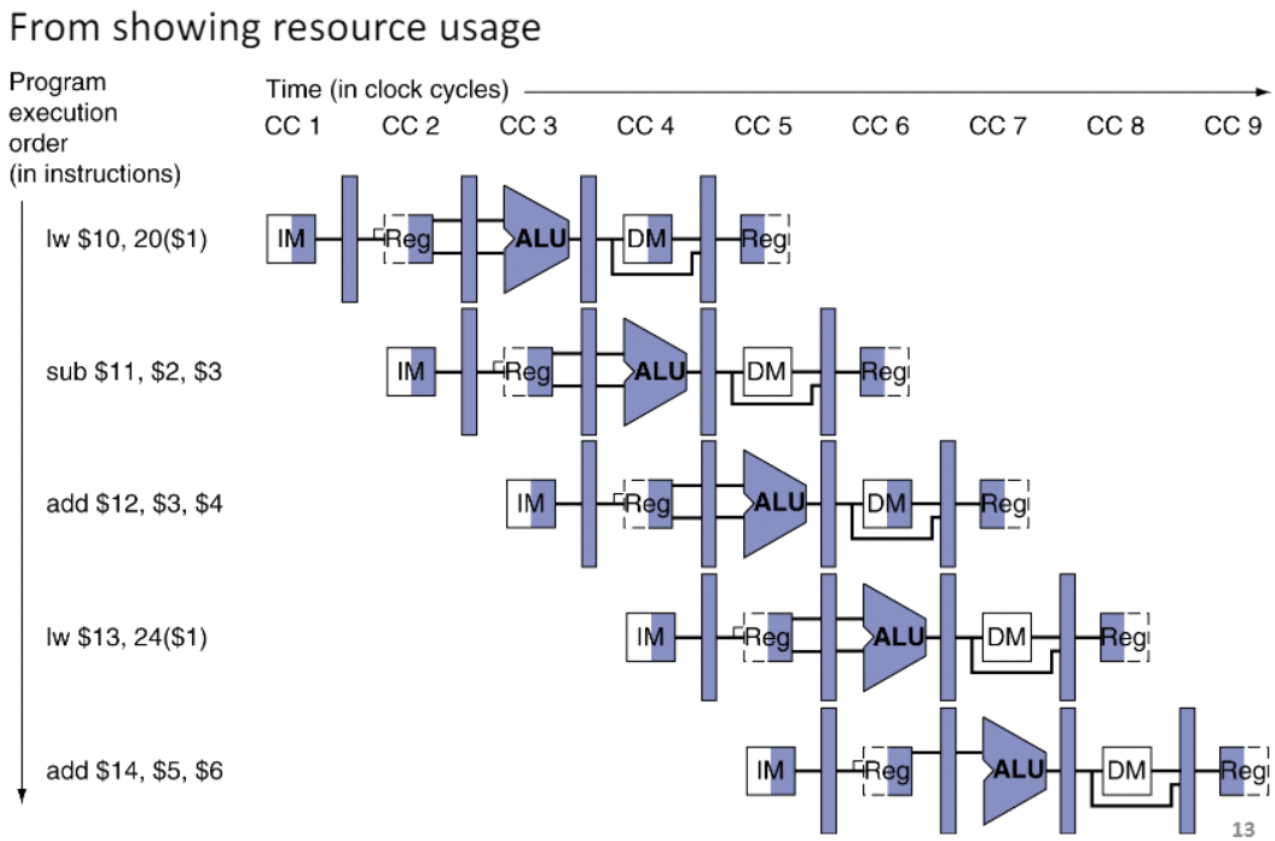
3-2. Multi-Cycle Microarchitecture

멀티 사이클(multi-cycle)은 컴퓨터 아키텍처에서 사용되는 명령어 처리 방법 중 하나이다. 이 방법은 명령어 처리 주기를 여러 개의 단계로 분할하는 대신, 하나의 주기 안에서 여러 동작을 수행하는 방식을 채택한다. 각 명령어는 다른 주기에 걸쳐 여러 단계를 거치며 처리된다.

한 클럭에 한 명령어를 실행해야 하는 Single-cycle processor에서는 모든 명령어가 5 단계를 거치는 것만큼의 처리 시간을 가져야 한다는 비효율이 있다. 메모리를 참조하고 그 값을 다시 레지스터에 저장하는 Load word는 위 5단계를 모두 거치고, 그 처리에 걸리는 시간이 곧 클럭 주기가 되어 3단계 만을 필요로 하는 명령어도 LW와 동일한 클럭 주기를 갖게 되는 것이다.

Multi-cycle 방식은 이 처리 stage 별로 클럭을 나눠 명령어를 처리한다. 이것으로 한 클럭 주기는 줄어들고, 3단계까지만 필요한 명령어는 3번의 클럭만, 5단계가 모두 다 필요한 명령어는 5번 클럭이 한 명령어 처리에 필요하게 되는 것이다. 이렇듯 멀티 사이클은 명령어의 종류에 따라 처리하는 데 걸리는 주기 수가 다를 수 있다. 각각의 주기는 특정 동작을 수행하는데 할당된다.

3-3. Pipelined Microarchitecture



파이프라인은 컴퓨터 아키텍처에서 사용되는 명령어 처리 기술 중 하나이다. 이 기술은 명령어

처리 주기를 여러 단계로 나누어 각각의 단계에서 병렬로 다른 명령어를 처리하는 방식을 채택한다. 명령어는 프로그램 순서대로 연속적인 단계에서 처리되며, 한 번에 하나의 명령어가 처리되는 것이 아니라 여러 명령어가 동시에 처리될 수 있다. 파이프라인은 주로 다음과 같은 단계로 구성된다.

IF: 메모리에서 다음 실행할 명령어를 가져온다.
ID: 명령어를 해석하고 필요한 데이터와 연산을 결정한다.
EX: 연산을 수행하고 결과를 계산한다.
MEM: 필요한 경우 메모리에 접근하여 데이터를 읽거나 쓰기도 한다.
WB: 최종 결과를 레지스터에 기록한다.

각 단계는 별도의 하드웨어 유닛으로 구성되며, 다른 명령어들이 각각의 단계를 거치며 동시에 처리된다. 이를 통해 병렬성을 활용하여 전체적인 처리 속도를 향상시킬 수 있다.

파이프라인은 처리 속도를 향상시키는 장점을 가지고 있지만, 명령어 간의 종속성과 분기 예측 등 몇 가지 문제에 대해서도 고려해야 한다. 데이터 종속성이 발생할 경우, 앞선 명령어의 결과를 기다려야 하는 대기 시간이 발생하며 이는 성능 저하를 가져올 수 있다. 또한, 분기 명령어의 경우, 분기 예측을 통해 올바른 명령어 경로를 선택하는 기능이 필요하다.

파이프라인은 현대 프로세서에서 널리 사용되며, 복잡한 명령어 집합과 다양한 기능을 처리하기 위해 여러 개의 파이프라인 단계를 가질 수도 있다. 이를 통해 프로세서의 성능을 극대화하고, 고속 연산과 동시성을 지원하는 효율적인 명령어 처리를 가능케 한다.

3-4. Hazard

컴퓨터 아키텍처에서 hazard는 파이프라인에서 발생할 수 있는 성능 저하나 오작동을 의미한다. 이러한 해저드는 명령어의 종속성 또는 분기 명령어의 예측 오류와 관련될 수 있다. 해저드는 세 가지 문제를 해결해야 한다.

구조적 해저드는 한 개 이상의 명령어를 중복된 하드웨어에서 처리하고자 하는 상황을 말한다. 예를 들면 fetch 단계와 memoryAccess단계에서 메모리 자원을 동시에 사용하고자 하는 경우를 말한다. 이런 경우 Bubble을 추가하여 중복 사용되는 하드웨어가 생기지 못하도록 하는 방식이 있을 수 있고, 또는 아예 자원을 추가하여 중복을 피할 수도 방법도 있다. 전통적인 5단계의 프로세서라면 명령어를 담는 메모리 자원과 데이터를 담는 메모리 자원을 분리하는 것으로 fetch 단계와 memoryAccess단계의 메모리 자원을 분리할 수 있게 되는 것이다.

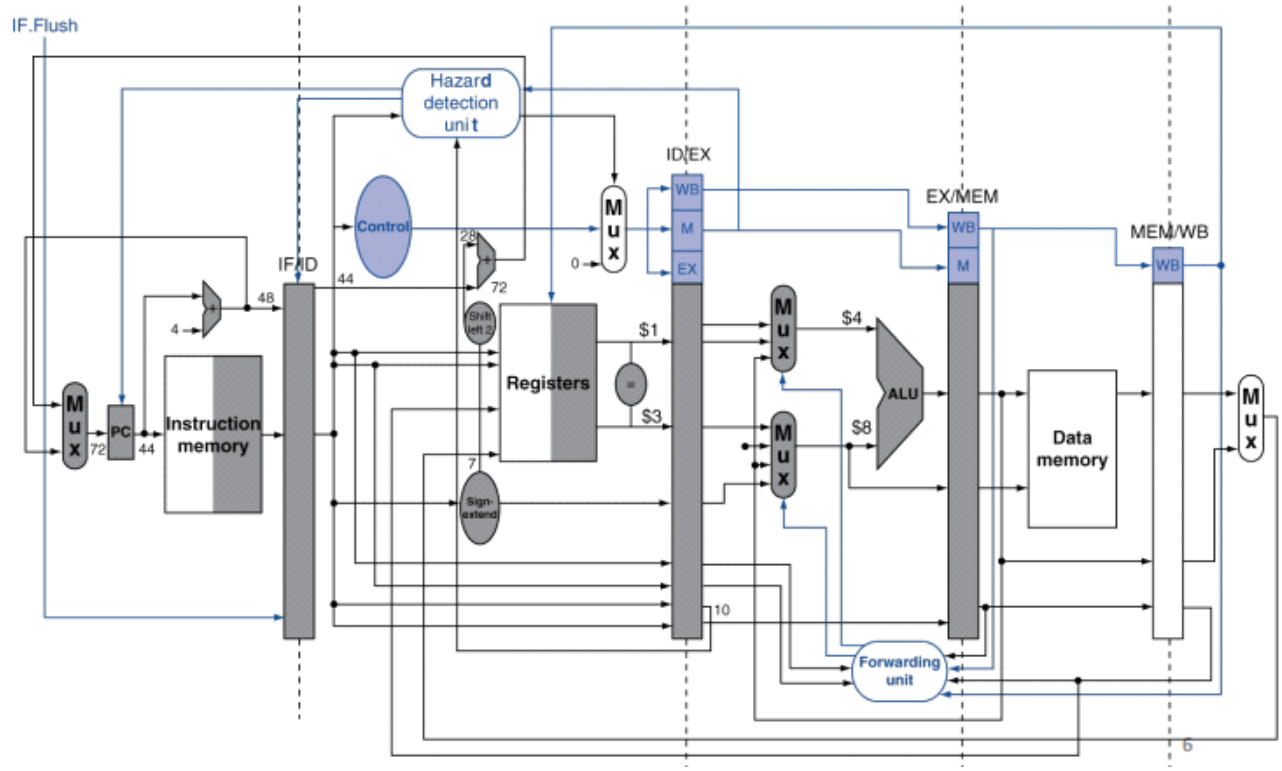
데이터 해저드는 명령어가 아직 처리 중인 앞선 명령어에 종속성을 가질 때 발생한다. 예를 들면 일렬의 명령어가 순차적으로 동일한 레지스터 R2를 쓰고, 읽는다고 가정해보자. 각각 execute, decode 단계에 있는 경우 decode 단계의 명령어는 앞선 명령어가 레지스터에 연산의 결과를 저장하지 않은 상태로 decode에서 그 값을 읽어버린다. execute 단계에 있는 '쓰기' 명령어가 writeBack 단계에 가서야 레스터에 값을 저장하기 때문이다. 따라서 이는 앞선 명령어가 writeBack를 처리하기까지 다음 명령어를 대기시키는 방식(Stalling 또는 Freezing), writeBack에 저장될 값이 결정되는 시점의 값을 미리 앞 당겨 사용하는 방식(Forwarding), 또는 아예 컴파일 시점에 컴파일러 수준에서 일정 거리 안에서 사용되는 레지스터의 중복 사용을 피하는 방식으로 해결할 수 있다.

제어 해저드는 분기 명령어의 결과를 알기 전까지 fetch된 명령어의 유효성 여부를 알 수 없는 문제가 발생한다. 명령어가 분기 명령인지 아닌지 또는 분기 명령어가 실행될지, 아닐지 모른다. 분기 명령이 실행되지 않음을 생각하여 우선 다음 pc의 명령어를 fetch 하나, 만일 분기가 맞다면 이를 알게되는 execute 단계 이전의 fetch, decode는 모두 무의미한 명령어가 수행된 것이다. 반대로 분기 명령어를 실행한다고 생각하였다가 execute 단계에서 분기가 아니라면, 마찬가지로 앞선 단계의 명령어는 무의미해진다.

이런 무의미한 명령어를 어떻게 줄이는가가 제어 해저드를 생각하는 주요 관점이 된다. 앞선 예시의 경우처럼 무의미한 명령어가 발생하면 이를 유효하지 않은 명령어로 처리하는 방법(Branch delay), 이전 분기 결과를 기억하는 등 분기 결과를 예측하는 방법(Branch prediction), 더 나아가 아예 PC와 분기 결과를 기록한 버퍼를 이용하여 decode 단계 이전에 분기 결과를 예측하는 방법 등을 사용한다.

3-5. Datapath

파이프라인의 데이터패스는 명령어가 파이프라인 스테이지를 통과하는 동안 데이터의 흐름을 관리하는 구성 요소이다. 데이터패스는 레지스터, ALU, 연산 장치 등 다양한 하드웨어 컴포넌트로 구성된다. 이러한 데이터패스 구성 요소들은 파이프라인의 각 스테이지에서 필요한 데이터를 전달하고 연산을 수행하는 역할을 한다. 명령어가 파이프라인을 통과하면서 필요한 데이터는 레지스터를 통해 전달되고, 알카인이나 메모리 등의 하드웨어 컴포넌트에서 연산과 데이터 액세스가 이루어진다. 데이터 선택기는 데이터의 종속성을 처리하거나 분기 예측을 위해 필요한 데이터를 선택하는 데 사용된다. 제어 유닛은 제어 신호를 생성하여 파이프라인의 동작을 조정하고 명령어의 흐름을 관리한다. 이렇게 구성된 데이터패스는 파이프라인의 성능을 향상시키고, 명령어 처리의 효율성을 극대화하는 데 기여한다.

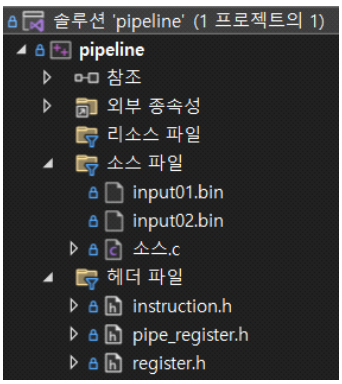


싱글사이클에서 각각의 단계 사이에 레지스터가 추가된 것을 볼 수 있다. 또한, 각각의 컨트롤 해저드, 데이터 해저드를 위해 하드웨어가 추가된 것을 볼 수 있다.

4. Implementation

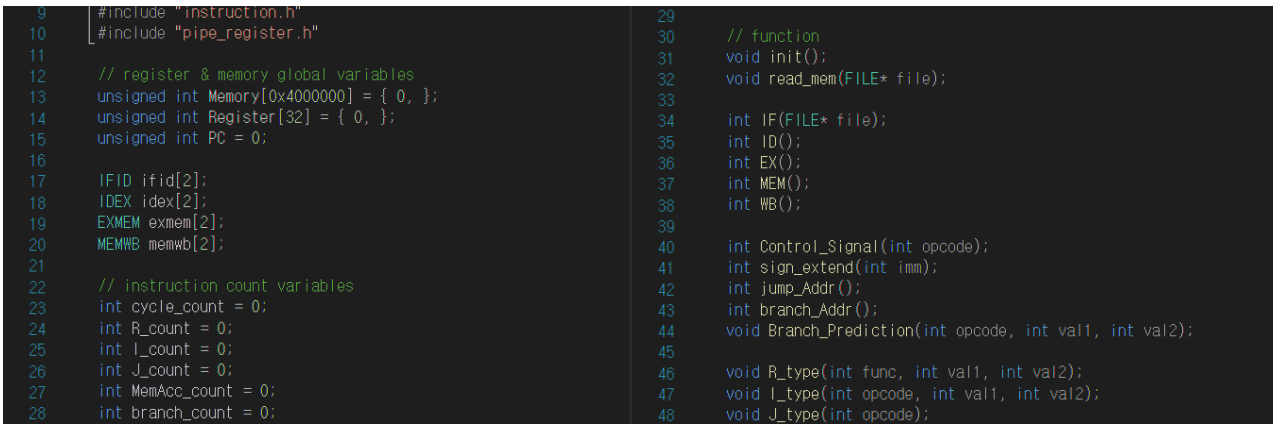
코드 설명에 들어가기에 앞서, 본 프로그램은 C언어로 코드를 짰다. C언어를 사용한 이유는 이전 과제인 싱글 사이클을 C언어로 구현한 것이 큰 이유이다. 지금부터 전체적인 구조와 함수 각각의 기능을 알아보도록 하겠다.

4-1. Program structure, global variables, functions



좌측 그림은 작성한 프로그램의 구조이다. instruction.h, register.h는 기존에 싱글 사이클 구현에서 사용했던 헤더 파일을 그대로 사용하였다. 참고로 instruction.h는 R타입의 함수 코드, I과 J 타입의 OP 코드가 매크로 형태로 정의되어 있으며, register.h는 각각의 레지스터 이름과 그에 맞는 레지스터 번호가 매크로 형태로 정의되어 있다. pipe_register.h는 각각의 단계에서 전달할 때 사용할 IFID, IDEX,

EXMEM, MEMWB의 레지스터를 구조체로 정의하였다. 구조체 안에서 선언된 변수는 opcode, rs, rd, rt, imm 등의 명령어, PC, 분기 시 사용될 주소, Control Unit 값 등이 담겨 있다.



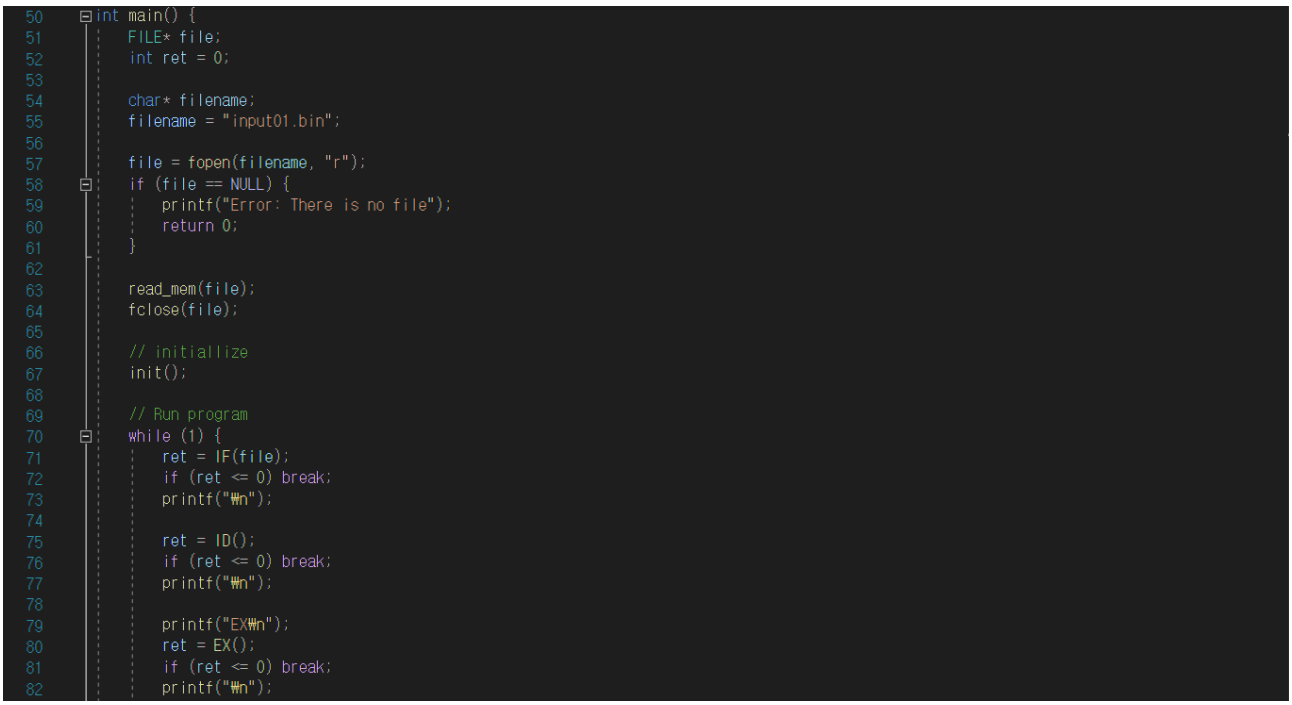
```
#include "instruction.h"
#include "pipe_register.h"

// register & memory global variables
unsigned int Memory[0x4000000] = { 0, };
unsigned int Register[32] = { 0, };
unsigned int PC = 0;

IFID ifid[2];
IDEX idex[2];
EXMEM exmem[2];
MEMWB memwb[2];

// instruction count variables
int cycle_count = 0;
int R_count = 0;
int I_count = 0;
int J_count = 0;
int MemAcc_count = 0;
int branch_count = 0;
```

```
// function
void init();
void read_mem(FILE* file);

int IF(FILE* file);
int ID();
int EX();
int MEM();
int WB();

int Control_Signal(int opcode);
int sign_extend(int imm);
int jump_Addr();
int branch_Addr();
void Branch_Prediction(int opcode, int val1, int val2);

void R_type(int func, int val1, int val2);
void I_type(int opcode, int val1, int val2);
void J_type(int opcode);
```

17 - 20 : 파이프라인 단계로 이동 시 값을 전달할 레지스터를 구조체 형식으로 선언했다. 인덱스가 [0]인 구조체는 새로운 값을 읽을 때 의미하며, [1]은 현재 단계에서 사용하는 기존의 값을 의미한다. 즉, 0은 앞 단계의 값이며 1은 뒤의 단계 값이다.

4-2. Main

다음 코드는 주어진 파일(“input01.bin”)에서 명령어를 읽어 실행하는 메인 함수이다. 명령어가 끝날 때까지 무한 루프로 설정되어 있다. 무한 루프 안에는 각각의 함수를 호출하여 명령어 단계를 실행한다. 프로그램이 종료되면 반환 값, 사이클 수를 표시한다.



```
int main() {
    FILE* file;
    int ret = 0;

    char* filename;
    filename = "input01.bin";

    file = fopen(filename, "r");
    if (file == NULL) {
        printf("Error: There is no file");
        return 0;
    }

    read_mem(file);
    fclose(file);

    // initiallize
    init();

    // Run program
    while (1) {
        ret = IF(file);
        if (ret <= 0) break;
        printf("₩n");

        ret = ID();
        if (ret <= 0) break;
        printf("₩n");

        printf("EX₩n");
        ret = EX();
        if (ret <= 0) break;
        printf("₩n");
```

54 - 64 : 파일 오픈 및 오류 처리, Memory에 저장

```
84          printf("MEM₩n");
85          ret = MEM();
86          if (ret <= 0) break;
87          printf("₩n");
88
89          printf("WB₩n");
90          ret = WB();
91          if (ret <= 0) break;
92          printf("₩n");
93
94          cycle_count++;
95          printf("₩n₩n");
96      }
97
98      printf("================================================================================₩n");
99      printf("Return value (r2)                  : %d₩n", Register[v0]);
100     printf("Total cycle                        : %d₩n", cycle_count);
101     printf("Executed 'R' instruction           : %d₩n", R_count);
102     printf("Executed 'I' instruction           : %d₩n", I_count);
103     printf("Executed 'J' instruction           : %d₩n", J_count);
104     printf("Number of branch taken             : %d₩n", branch_count);
105     printf("Number of memory access instructions : %d₩n", MemAcc_count);
106     printf("================================================================================₩n");
107
108     return 0;
109 }
```

70 - 96 : 프로그램 실행

98 - 106 : 결과 출력

4-3. Init, Read_mem

초기화 작업을 수행하는 init 함수와 파일에서 데이터를 읽어와 메모리에 저장하는 read_mem 함수를 포함하고 있다. init 함수는 RA 레지스터와 스택 포인터를 초기화하고, read_mem 함수는 파일로부터 데이터를 읽어와서 메모리에 저장한다.

```
111 void init() {
112     Register[ra] = 0xFFFFFFFF;
113     Register[sp] = 0x10000000;
114 }
```

112 : Return Address를 0xFFFFFF으로 초기화

113 : Stack Pointer를 0x10000000으로 초기화

```
116 void read_mem(FILE* file) {
117     unsigned int ret = 0;
118     int i = 0;
119     int memVal;
120
121     while (true) {
122         int data = 0;                                   // 파일을 읽어오는 용도로 사용할 변수
123         ret = fread(&data, 1, sizeof(int), file);       // data 변수에 파일로부터 4바이트 만큼 데이터를 읽어옴. ret 변수에는 fread 함수의 결
124         memVal = ((data & 0xFF000000) >> 24) |
125             ((data & 0x00FF0000) >> 8) |
126             ((data & 0x0000FF00) << 8) |
127             ((data & 0x000000FF) << 24);
128         Memory[i++] = memVal;                           // Memory에 값을 저장
129
130
131         if (ret != 4) break;                            // 4바이트를 읽지 못했다면 루프 탈출
132     }
133 }
```

123 : file에서 4바이트 데이터를 읽어온 후, ret에 fread()가 읽은 바이트 수 저장.

124 - 127 : data 값을 비트 연산을 사용하여 엔디안 변환을 수행한 후 memVal 변수에 저장.

128 : Memory 배열에 memVal 값을 저장.

131 : ret 변수가 4가 아니라면(즉, 4바이트를 읽지 못했다면) 루프를 탈출.

4-4. IF

파이프라인의 첫 번째 스테이지인 "IF (Instruction Fetch)"를 구현한 부분이다. 주어진 파일 포인터를 통해 명령어를 읽어와 파이프라인의 IF 스테이지를 수행한다. IF 함수는 파일 포인터를 매개변수로 받고, 정수형 값을 반환하는 함수이다. 파일에서 명령어를 읽어와 IF 스테이지를 수행하고, IF 스테이지의 성공 여부를 반환한다.

```
135  int IF(FILE* file) {
136      int ret = 1;
137      if (PC == 0xFFFFFFFF) return 0;                        // PC가 0xFFFFFFFF이면 종료
138      ifid[0].inst = Memory[PC / 4];
139      ifid[0].PC = PC;
140
141      printf("Cycle[%03d]============================================================₩n₩n", cycle_count + 1);
142      printf("[IF]₩n");
143
144      if (ifid[0].inst == 0) {
145          PC += 4;
146          cycle_count++;
147          printf("Bubble₩n");
148          return IF(file);  // nop 명령어일 경우 다시 fetch 호출
149      }
150
151      PC += 4;
152
153      printf("₩t[IM] PC : 0x%X -> 0x%08X₩n", ifid[0].PC, ifid[0].inst);
154      printf("₩t[PC UPDATE] PC <- 0x%08X = 0x%08X + 4₩n", PC, ifid[0].PC);
155
156      return ret;
157  }
```

137 : PC 값이 0xFFFFFFFF인 경우 함수를 종료하고 0을 반환

138 : PC 값을 기반으로 메모리에서 다음 실행할 명령어를 가져와 저장

144 - 149 : 가져온 명령어가 nop인 경우, IF를 다시 호출

151 : 다음 명령어를 가져오기 위해 PC 값을 4만큼 증가

156 : 함수가 성공적으로 실행되었음을 반환

4-5. ID

파이프라인의 두 번째 스테이지인 "ID (Instruction Decode)"를 구현한 부분이다. 해당 스테이지에서는 IF 스테이지에서 가져온 명령어를 디코딩하고, 필요한 정보를 추출한다. ID 함수는 정수형 값을 반환하는 함수이다. ID 스테이지의 작업을 수행하고, 스테이지의 성공 여부를 나타내는 값을 반환한다. 각각의 명령어를 추출하기 위해 비트 마스킹과 시프트 연산을 사용하며, 명령어의 유형에 따라 적절한 출력을 수행한다. 예를 들어, R 타입 명령어인 경우 레지스터 정보와 함수 코드를 출력하고, J 타입 또는 JAL 타입 명령어인 경우 분기 주소를 출력하며, 그 외의 경우 I 타입 명령어로 간주하여 레지스터 정보와 즉시값을 출력한다.

```
222  int ID() {
223      int ret = 1;
224      int opcode;
225
226      ifid[1] = ifid[0];
227
228      idex[0].opcode = (ifid[1].inst & 0xFC000000);
229      idex[0].opcode = (idex[0].opcode >> 26) & 0x3F;
230      opcode = idex[0].opcode;
231
232      idex[0].rs = (ifid[1].inst & 0x03E00000);
233      idex[0].rs = (idex[0].rs >> 21) & 0x1F;
234
235      idex[0].rt = (ifid[1].inst & 0x001F0000);
236      idex[0].rt = (idex[0].rt >> 16) & 0x1F;
237
238      idex[0].rd = (ifid[1].inst & 0x0000F800);
239      idex[0].rd = (idex[0].rd >> 11) & 0x1F;
240
241      idex[0].shamt = (ifid[1].inst & 0x000007C0);
242      idex[0].shamt = idex[0].shamt >> 6;
243
244      idex[0].func = (ifid[1].inst & 0x0000003F);
245
246      idex[0].imm = (ifid[1].inst & 0x0000FFFF);
247
248      idex[0].addr = (ifid[1].inst & 0x03FFFFFF);
249
250      printf("[ID]₩n");
251
252      switch (opcode) {
253      case 0:
254          printf("₩t[TYPE : R]    OPCODE : 0x%X, RS : 0x%X (R[%d] = 0x%X), RT : 0x%X (R[%d] = 0x%X), RD : 0x%X (R[%d] = 0x%X), SHAMT : 0x%X, F
255
256          R_count++;
257          break;
258
259      case J:
260          printf("₩t[TYPE : J]    OPCODE : 0x%X, BRANCH : 0x%X₩n", opcode, idex[0].addr);
261
262          J_count++;
263          break;
264
265      case JAL:
266          printf("₩t[TYPE : J]    OPCODE : 0x%X, BRANCH : 0x%X₩n", opcode, idex[0].addr);
267
268          J_count++;
269          break;
270
271      default:
272          printf("₩t[TYPE : I]    OPCODE : 0x%X, RS : 0x%X (R[%d] = 0x%X), RT : 0x%X (R[%d] = 0x%X), IMM : 0x%X₩n", opcode, idex[0].rs, idex[0
273
274          I_count++;
275          break;
276      }
277
278      int imm = ifid[1].inst;
279      idex[0].SignExtImm = sign_extend(imm);
280
281      idex[0].inst = ifid[1].inst;
282      idex[0].PC = ifid[1].PC;
283
284      J_type(opcode);
285      Control_Signal(opcode);
286
287      return ret;
288  }
```

226 : IF 스테이지에서 가져온 명령어와 PC 값을 다음 스테이지인 ID/EX 레지스터에 복사

228 - 248 : 명령어를 디코딩하여 필요한 정보를 추출

252 - 276 : : 추출된 opcode 값을 기반으로 명령어의 유형에 따라 적절한 출력 수행 및 카운터 변수 증가

279 : 디코딩된 명령어의 imm 값을 sign_extend 함수를 이용하여 부호 확장

281 : 디코딩된 명령어를 ID/EX 레지스터에 저장

282 : 이전 스테이지에서 전달된 PC 값을 ID/EX 레지스터에 저장

284 : J 타입 명령어인 경우 J_type 함수를 호출하여 해당 명령어에 대한 처리를 수행

285 : opcode 값을 기반으로 제어 신호를 설정하는 Control_Signal 함수를 호출

287 : 함수가 성공적으로 실행되었음을 반환

4-6. EX

파이프라인의 세 번째 스테이지인 "EX (Execute)"를 구현한 부분이다. EX 스테이지에서는 디코딩된 명령어에 기반하여 연산을 수행하고 결과를 계산한다. EX 함수는 정수형 값을 반환하는 함수이다. EX 스테이지의 작업을 수행하고, 스테이지의 성공 여부를 나타내는 값을 반환한다.

```
548  int EX() {
549      int ret = 1;
550      int val1, val2;
551
552      idex[0].val1 = Register[idex[0].rs];
553      idex[0].val2 = Register[idex[0].rt];
554
555      idex[1] = idex[0];
556
557      idex[1].ZeroExtImm = (idex[1].inst & 0x0000FFFF);
558
559      // data forwarding
560      if ((idex[1].rs != 0) && (idex[1].rs == exmem[0].WriteReg) && (exmem[0].RegWrite)) {
561          idex[1].val1 = exmem[0].ALUResult;
562      }
563      else if ((idex[1].rs != 0) && (idex[1].rs == memwb[0].WriteReg) && (memwb[0].RegWrite)) {
564          if (memwb[0].MemtoReg == 1) {
565              idex[1].val1 = memwb[0].ReadData;
566          }
567          else {
568              idex[1].val1 = memwb[0].ALUResult;
569          }
570      }
571      if ((idex[1].rt != 0) && (idex[1].rt == exmem[0].WriteReg) && (exmem[0].RegWrite)) {
572          idex[1].val2 = exmem[0].ALUResult;
573      }
574      else if ((idex[1].rt != 0) && (idex[1].rt == memwb[0].WriteReg) && (memwb[0].RegWrite)) {
575          if (memwb[1].MemtoReg == 1) {
576              idex[1].val2 = memwb[0].ReadData;
577          }
578          else {
579              idex[1].val2 = memwb[0].ALUResult;
580          }
581      }
582
583      val1 = idex[1].val1;
584      val2 = idex[1].val2;
585
586      if (idex[1].RegDst == 1) {
587          R_type(idex[1].func, val1, val2);
588          R_count++;
589      }
590      else if (idex[1].ALUSrc == 1) {
591          I_type(idex[1].opcode, val1, val2);
592      }
593      else if (idex[1].Branch == 1) {
594          Branch_Prediction(idex[1].opcode, val1, val2);
595      }
596
597      if (idex[1].RegDst == 1) { // R-type
598          exmem[0].WriteReg = idex[1].rd;
599      }
600      else {
601          exmem[0].WriteReg = idex[1].rt;
602      }
603
```

552 - 553 : 필드에 해당하는 레지스터에서 값을 읽어와 저장

555 : idex[0]의 값을 idex[1]에 복사

557 : 명령어의 하위 16비트를 ZeroExtImm 필드에 저장

560 - 581 : 데이터 forwarding을 통해 앞선 스테이지(exmem, memwb)에서 필요한 값을 가져온다. 데이터가 forwarding될 수 있는 경우 해당 값을 가져와 val1 또는 val2에 할당한다.

586 - 595 : RegDst 신호에 따라 R 타입, ALUSrc 신호에 따라 I 타입, Branch 신호에 따라 분기 처리를 수행하는 함수를 호출. 해당 함수들은 해당하는 연산을 수행하고 결과를 반환

597 - 602 : RegDst 신호에 따라 exmem[0].WriteReg 값을 idex[1].rd 또는 idex[1].rt로 설정

```
604     // Update Latch : exmem[0] = idex[1]
605     exmem[0].opcode = idex[1].opcode;
606     exmem[0].func = idex[1].func;
607     exmem[0].inst = idex[1].inst;
608     exmem[0].PC = idex[1].PC;
609     exmem[0].Branch = idex[1].Branch;
610     exmem[0].MemRead = idex[1].MemRead;
611     exmem[0].MemtoReg = idex[1].MemtoReg;
612     exmem[0].MemWrite = idex[1].MemWrite;
613     exmem[0].RegWrite = idex[1].RegWrite;
614     exmem[0].val1 = val1;
615     exmem[0].val2 = val2;
616
617     return ret;
618 }
```

605 - 615 : exmem[0] 레지스터에 idex[1]의 필드 값을 복사 (업데이트)

617 : 함수가 성공적으로 실행되었음을 반환

4-7. MEM

파이프라인의 네 번째 스테이지인 "MEM (Memory Access)"를 구현한 부분이다. MEM 스테이지에서는 메모리 액세스가 필요한 경우 메모리에 데이터를 읽거나 쓰는 작업을 수행한다.

```
620 int MEM() {
621     int ret = 1;
622
623     exmem[1] = exmem[0];
624
625     if (exmem[1].MemRead == 1) { // LW
626         exmem[1].ReadData = Memory[exmem[1].temp / 4];
627
628         printf("₩tR[%d] <- 0x%X₩n", exmem[1].rs, exmem[1].ReadData);
629
630         MemAcc_count++;
631     }
632     else if (exmem[1].MemWrite == 1) { // SW
633         Memory[exmem[1].temp / 4] = exmem[1].val2;
634
635         printf("₩tR[%d] -> 0x%X₩n", exmem[1].rs, exmem[1].val2);
636
637         MemAcc_count++;
638     }
639
640     // Update Latch : memwb[0] = exmem[1]
641     memwb[0].opcode = exmem[1].opcode;
642     memwb[0].func = exmem[1].func;
643     memwb[0].inst = exmem[1].inst;
644     memwb[0].MemtoReg = exmem[1].MemtoReg;
645     memwb[0].RegWrite = exmem[1].RegWrite;
646     memwb[0].val1 = exmem[1].val1;
647     memwb[0].val2 = exmem[1].val2;
648     memwb[0].ALUResult = exmem[1].ALUResult;
649     memwb[0].WriteReg = exmem[1].WriteReg;
650     memwb[0].ReadData = exmem[1].ReadData;
651
652     return ret;
653 }
```

623 : exmem[0]의 값을 exmem[1]에 복사

625 - 631 : MemRead가 활성화된 경우, 메모리에서 데이터를 읽어와 exmem[1].ReadData에 저장

632 - 638 : MemWrite가 활성화된 경우, exmem[1].val2의 값을 메모리에 저장

641 - 650 : memwb[0] 레지스터에 exmem[1]의 필드 값을 복사

652 : 함수가 성공적으로 실행되었음을 반환

4-8. WB

파이프라인의 마지막 스테이지인 "WB (Write Back)"를 구현한 부분이다. 이 코드는 파이프라인의 WB 스테이지에서 레지스터에 데이터를 쓰는 작업을 수행한다. WB 스테이지는 최종 스테이지로, 결과를 레지스터에 저장하여 프로그램 실행의 최종 결과를 반영한다.

```
655  int WB() {
656      int ret = 1;
657
658      memwb[1] = memwb[0];
659
660      if (memwb[1].RegWrite == 1) {
661          if (memwb[1].MemtoReg == 1) {
662              Register[memwb[1].WriteReg] = memwb[1].ReadData;
663          }
664          else {
665              if (memwb[1].WriteReg != 0) {
666                  Register[memwb[1].WriteReg] = memwb[1].ALUResult;
667              }
668              else {
669                  Register[memwb[1].WriteReg] = 0;
670              }
671          }
672      }
673
674      printf("₩tR[%d] <- 0x%X₩n", memwb[1].WriteReg, Register[memwb[1].WriteReg]);
675
676      return ret;
677  }
```

658 : memwb[0]의 값을 memwb[1]에 복사

660 - 672 : RegWrite가 활성화된 경우, MemtoReg 플래그에 따라 메모리에서 읽은 데이터 또는 ALU의 결과를 레지스터에 저장. 만약 WriteReg가 0인 경우, 레지스터 0에 값을 저장

676 : 함수가 성공적으로 실행되었음을 반환

4-9. Control_Signal

제어 신호를 설정하는 부분을 구현한 함수이다. 해당 함수는 파이프라인의 ID 스테이지에서 실행되며, 입력된 opcode를 기반으로 제어 신호를 설정한다. 이 코드는 ID 스테이지에서 현재 명령어의 opcode를 기반으로 제어 신호를 설정한다. 제어 신호는 파이프라인의 다른 스테이지에서 해당 동작을 수행하도록 지시하는 역할을 한다.

```
159  int Control_Signal(int opcode) {
160      int ret = 1;
161
162      // MUX
163      if (opcode == 0x0) {
164          idex[0].RegDst = 1;
165      }
166      else {
167          idex[0].RegDst = 0;
168      }
169
170      if ((opcode == J) || (opcode == JAL) || (opcode == BEQ) || (opcode == BNE)) {
171          idex[0].Branch = 1;
172      }
173      else {
174          idex[0].Branch = 0;
175      }
176
177      if (opcode == LW) {
178          idex[0].MemtoReg = 1;
179          idex[0].MemRead = 1;
180      }
181      else {
182          idex[0].MemtoReg = 0;
183          idex[0].MemRead = 0;
184      }
185
186      if (opcode == SW) {
187          idex[0].MemWrite = 1;
188      }
189      else {
190          idex[0].MemWrite = 0;
191      }
192
193      if ((opcode != 0) && (opcode != BEQ) && (opcode != BNE)) {
194          idex[0].ALUSrc = 1;
195      }
196      else {
197          idex[0].ALUSrc = 0;
198      }
199
200      if ((opcode != SW) && (opcode != BEQ) && (opcode != BNE) && (opcode != JR) && (opcode != J) && (opcode != JAL)) {
201          idex[0].RegWrite = 1;
202      }
203      else {
204          idex[0].RegWrite = 0;
205      }
206
207      return ret;
208  }
```

163 - 168 : opcode가 0x0 (R-type)인 경우, RegDst를 1로 설정하여 rd 필드를 대상으로 레지스터를 선택

170 - 175 : opcode가 J, JAL, BEQ, BNE 중 하나인 경우, Branch를 1로 설정하여 분기 동작을 활성화

177 - 184 : opcode가 LW인 경우, MemtoReg를 1로 설정하여 메모리에서 읽은 데이터를 레지스터에 전달. 동시에 MemRead도 1로 설정하여 메모리 읽기 동작을 활성화

186 - 191 : opcode가 SW인 경우, MemWrite를 1로 설정하여 메모리 쓰기 동작을 활성화

193 - 198 : opcode가 R-type이 아니고, BEQ, BNE도 아닌 경우, ALUSrc를 1로 설정하여 rt 필드 값을 ALU의 두 번째 입력으로 선택. 그렇지 않은 경우, ALUSrc를 0으로 설정하여 imm 값을 ALU의 두 번째 입력으로 선택

200 - 205 : opcode가 SW, BEQ, BNE, JR, J, JAL이 아닌 경우, RegWrite를 1로 설정하여 레지스터 쓰기 동작을 활성화

207 : 함수가 성공적으로 실행되었음을 반환

4-10. Sign_extend

주어진 immediate 값을 sign-extend하여 반환하는 함수이다. 이 함수는 MIPS 아키텍처의 명령어 처리에서 immediate 값을 sign-extend하여 사용하는 데에 활용된다. 주로 분기 명령어나 데이터 전송 명령어에서 immediate 값을 확장하여 사용한다. sign-extend는 주어진 immediate 값을 부호 확장하여 해당 값을 더 큰 자료형으로 표현하는 작업을 수행한다.

```
210  int sign_extend(int imm) {
211      int sign_imm;
212      int sign = (imm & 0x0000FFFF) >> 15;
213      if (sign == 1) {
214          sign_imm = 0xFFFF0000 | imm;
215      }
216      else {
217          sign_imm = 0x0000FFFF & imm;
218      }
219      return sign_imm;
220  }
```

212 : imm 값의 최상위 비트를 추출. 이를 위해 imm 값과 0x0000FFFF를 AND 연산하여 16비트 이상의 비트를 제거하고, 15번 비트를 추출한 후 오른쪽으로 15비트 이동

213 - 215 : 추출한 sign이 1인 경우, sign_imm 변수에 imm 값을 sign-extend하여 저장

216 - 218 : 추출한 sign이 0인 경우, sign_imm 변수에 imm 값을 그대로 저장

4-11. Jump_Addr, Branch_Addr

jump_Addr 함수는 점프 명령어(J 또는 JAL)의 목적 주소를 계산하여 반환하는 함수이며, branch_Addr 함수는 분기 명령어(BEQ 또는 BNE)의 분기 주소를 계산하여 반환하는 함수이다. 이러한 함수들은 MIPS 아키텍처의 점프(J, JAL)와 분기(BEQ, BNE) 명령어에서 목적 주소와 분기 주소를 계산하는 데에 사용된다. 계산된 주소는 명령어의 흐름을 변경하거나 프로그램 카운터(PC)를 갱신하는 데에 활용된다.

```
291  int jump_Addr() {
292      idex[0].JumpAddr = (ifid[1].PC & 0xF0000000) | (idex[0].addr << 2);
293      int j = idex[0].JumpAddr;
294      return j;
295  }
```

292 : 현재 PC의 상위 4비트와 왼쪽으로 2비트 시프트하여 계산된 주소를 OR 연산

294 : 함수가 정상적으로 실행되었을 경우, J 값을 반환

```
297  int branch_Addr() {
298      idex[1].BranchAddr = idex[1].SignExtImm << 2;
299      int b = idex[1].BranchAddr;
300      return b;
301  }
```

298 : idex[1] 구조체의 SignExtImm 멤버를 왼쪽으로 2비트 시프트

300 : 함수가 정상적으로 실행되었을 경우, B 값을 반환

4-12. Branch_Prediction

Branch_Prediction 함수로, 분기(BEQ 또는 BNE) 명령어의 조건을 평가하여 분기가 발생할 경우에 대한 처리를 수행한다. 이 함수는 분기 명령어의 조건을 평가하여 분기가 발생하는 경우에는 분기 주소를 계산하고, PC를 업데이트하여 프로그램의 흐름을 변경한다. 또한, 파이프라인 스테이지의 내용을 초기화하여 분기에 따른 명령어 플러시를 수행한다.

```
515  void Branch_Prediction(int opcode, int val1, int val2) {
516      switch (opcode) {
517      case BEQ:
518          if (val1 == val2) {
519              int b = branch_Addr();
520              PC = idex[1].PC + b + 4;
521              //flush
522              memset(&ifid[0], 0, sizeof(IFID));
523              memset(&idex[0], 0, sizeof(IDEX));
524
525              branch_count++;
526              printf("WtRS : 0x%X (R[%d] = 0x%X), RT : 0x%X (R[%d] = 0x%X), PC = 0x%X", idex[1].rs, idex[1].rs, val1, idex[1].rt, idex[1].rt,
527          }
528          break;
529
530      case BNE:
531          if (val1 != val2) {
532              int b = branch_Addr();
533              PC = idex[1].PC + b + 4;
534              //flush
535              memset(&ifid[0], 0, sizeof(IFID));
536              memset(&idex[0], 0, sizeof(IDEX));
537
538              branch_count++;
539              printf("WtRS : 0x%X (R[%d] = 0x%X), RT : 0x%X (R[%d] = 0x%X), PC = 0x%X", idex[1].rs, idex[1].rs, val1, idex[1].rt, idex[1].rt,
540          }
541          break;
542
543      default:
544          break;
545      }
546  }
```

518 : 만약 val1과 val2가 같다면, 분기 조건이 충족되었으므로 분기 주소(b)를 계산

520 : 다음 명령어의 주소를 계산하여 PC(Program Counter)에 저장

522 - 523 : 파이프라인 스테이지(IFID, IDEX)의 내용을 초기화하여 이전 단계의 명령어들을 플러시

530 - 541 : 위의 방식과 같으므로 설명 생략

5. Result

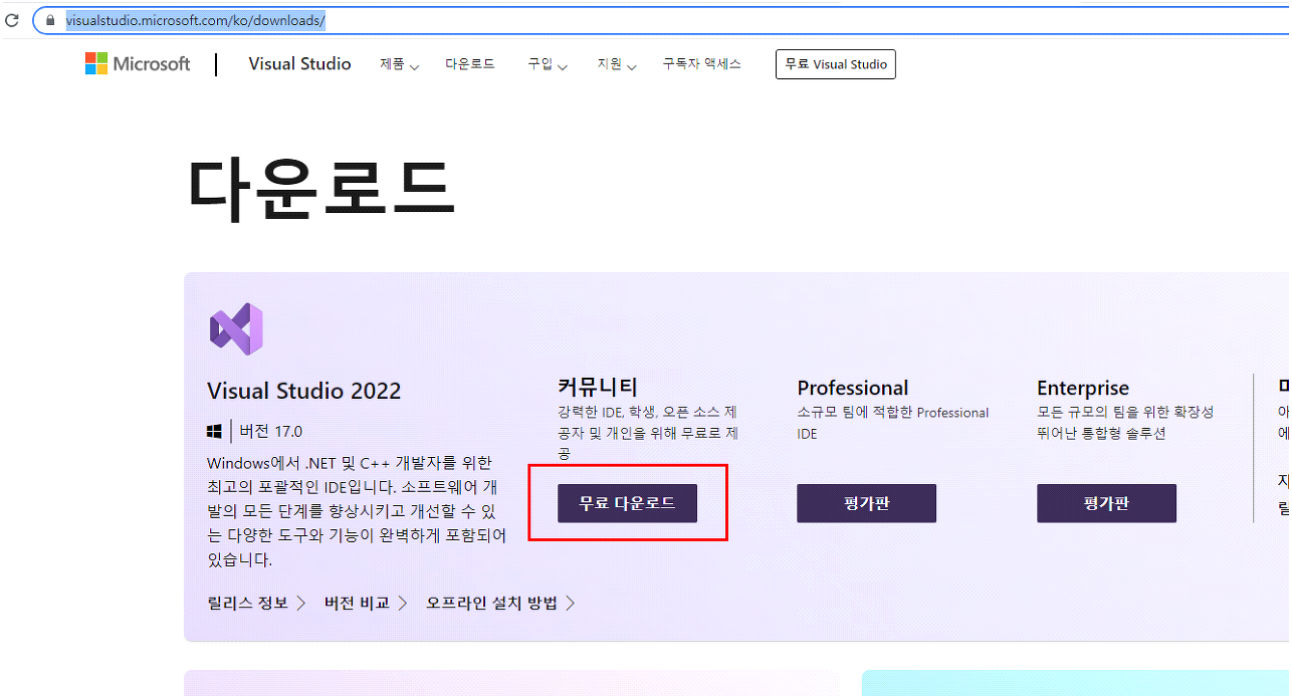
위에서 본 코드를 돌리면 어떻게 출력이 되는지 알아보도록 하겠다.


보는 것처럼 중간중간 사이클을 확인했을 때 모든 단계에서 잘 수행되는 것을 볼 수 있으며, 결과도 깔끔하게 나온 것을 볼 수 있다.

6. Build Environment

본 노트북의 운영체제는 윈도우 10이고, 시스템은 64비트 운영체제이다. 프로그램은 Visual Studio 2022를 사용하였으며, 설치방법은 다음과 같다.

1. 다운로드 사이트 접속 후 다운로드



2. 설치할 항목 선택 후 설치 진행 : 내려받은 파일을 실행하면 설치할 항목들이 나타난다. 그 중, 맞는 항목을 선택 후 설치한다.

3. 이후 설치가 완료되면, 로그인하면 된다. 로그인 계정을 바로 사용하지 않을 때는 나중에 로그인 항목을 선택한다.

다음은 Visual Studio에서 프로그램을 실행하는 방법에 관해서 설명하겠다.

1. 해당 압축 폴더를 압축을 풀고, Visual Studio를 열어서 [새프로젝트 만들기] - [빈프로젝트]를 클릭한다.
2. Windows에서 프로젝트 폴더를 열어 c 파일과 바이너리 파일을 넣어준다. 바이너리 파일과 c

파일이 같은 위치에 있어야 함을 유의한다.
3. 다시 생성한 프로젝트에 들어가 상단에 있는 [로컬 Windows 디버거]를 클릭하여 프로그램을 실행한다. 위와 같은 방법으로 파일을 실행하면, 디버그 콘솔에서 출력물을 볼 수 있다.

7. Lesson

이렇게 MIPS 아키텍처를 기반으로 한 파이프라인 시뮬레이터를 구현하였다. 주어진 입력 파일을 읽고, 명령어를 실행하여 레지스터와 메모리 값을 조작하며 프로그램을 실행한다. 이 프로그램은 R 타입, I 타입, J 타입의 명령어를 처리할 수 있다. 명령어 실행 도중에는 다양한 Control 신호와 MUX를 사용하여 연산 및 제어를 진행하였다. 프로그램의 실행 결과로는 레지스터의 최종 값을 출력하고 실행한 명령어의 종류와 수, 분기 횟수, 메모리 접근 횟수 등의 정보를 표시한다.

과제를 할 때, 처음에 싱글 사이클을 조금만 손보면 되겠다는 생각으로 쉽게 생각하였다. 하지만 병렬 처리 과정을 어떻게 해야 할지, 해저드를 어떻게 처리해야 할지 고민을 많이 했던 것 같다. 과제를 하면서 컴퓨터 구조론 책에서 영감을 받아서 코딩한 것이 조금 많았다. 예를 들면, 병렬 처리를 할 때, 레지스터 읽기와 쓰기를 나눠서 하는 법이 있었다. 과제를 하면서 정확히 이해하지 못했던 데이터 패스를 알 수 있게 되었던 것 같다.